Overview/Description

The project our group developed this semester, titled CrowdSauce, is a social networking application that allows end-users to share and discuss recipes. In many other popular social networking platforms such as Snapchat and Instagram, many end-users use the functionality to post pictures of their food. CrowdSauce seeks to build upon the core idea that users like sharing their food by providing an application that allows users to post pictures of the food they are eating along with a recipe describing how to make it. We see many use cases for this platform notably, college students who are living on a budget have access to all of their friends food creations and how they made them, users sharing satirical images of food that didn't quite work out, or users sharing their favorite home-recipes with all their friends. Succinctly put, CrowdSauce is a social networking application providing users a platform for sharing and finding recipes from their friends.

Software Development Process

For our project this semester, our team adopted a development style that used certain aspects of the Extreme Programming (XP) system, and drop others. One aspect of XP that we used was the pre-development usage of User Stories to define specifications. Similar to XP's Planning Game, the team broke down the project's goals into iterations based on user stories, and tasks necessary to complete the user stories. By first starting with user stories, our team started on the same page immediately. Another aspect of XP we adopted was the rapid development of our user stories and the consistent use of refactoring. Faster development was not only more exciting, but allowed for discovering if tasks were more technically difficult than we had imagined, and/or if certain specifications needed to be updated. With a 'make it work' first mentality, XP lead to consistent refactoring as more features were added onto the codebase. The refactoring was never too difficult, however. Refactoring processes were also taught during the course, so we were all prepared to update the codebase properly. Agreeing on the same coding standards and design practices also helped our team become a more consolidated group of programmers. The result was a much cleaner, unified codebase.

While there are portions of the XP development system that we used in our project, our development style differed from XP in a few ways. For CS427, we not only used pair programming but also had pairs constantly rotate and place on different iteration tasks.While we did rotate and each programmer touched the entirety of the code stack, it naturally occurred that certain individuals became the 'go-to' people for certain aspects of the code base. Because the CS427 teams ended up naturally divided, we decided that our group this semester will not be rotating the developers completely up/down the technology stack. It did not seem like the best use of everyone's time if after each Iteration new students have to learn a component of the codebase, as opposed to continuing to work on the technologies that they feel they have come to understand best. Instead we decided to split the team into two subteams – Front End and Back End. The divide allowed all of us to focus on one portion of our code stack. By staying on one subteam for the duration of the project, we were able to familiarize ourselves with the technologies and processes associated with tools being used (e.g. React.js vs. RethinkDB), and more rapidly develop the iteration requirements. There was less time spent learning how to use new tools, and less time lost or confused. The development style seemed to be an effective usage of our team's skills and time. By adhering to certain portions of the XP system and amending others, we were able to effectively carry out our project development efficiently.

Requirements & Specifications

The requirements and specifications for this project are derived from the several user stories and use cases that were implemented during development. Details of these user stories and use cases are given below:

User can login through Facebook	Users are required to have a Facebook account to join CrowdSauce. Upon reaching our website, users are prompted to login using their Facebook credentials, which is authenticated using Facebook's login authentication API. Once

	users log in for the first time, their details are collected using the Facebook API and are then stored.
User can post recipes	Users can post a recipe using a form that includes the following fields: title, ingredients list, directions (if personal recipe) or link to recipe (if recipe is from another website), images, notes, rating, preparation time, and difficulty level.
User can edit/delete recipes	The fields of a post can be edited by the user who made the post. Also, the post can be deleted by the user.
User can find past posts through tagging and search	When creating a recipe post, users can add tags to the post. These are one or two word descriptions of the meal that is prepared, and they can be tags that have already been used by other users on the website or even new tags they choose to introduce. Users can search for a particular meal they may be interested in using the search bar at the top of the feed page or by clicking a tag on a post. The search returns recipe posts made by users' friends that have tags that correspond to the search query. Multiple tags can be searched for at the same time in the search bar, and the search returns posts ordered by relevancy. In addition, the user can use the search bar to filter out posts by rating and order them by the number of "favorites".
User can view friends list	Users can view a list of CrowdSauce users who happen to be their friends on Facebook on feed page.
User can view a feed of recipe posts	Upon logging in, users are taken to a feed page that displays their own recipe along with the friends' recipe posts. The feed is personalized to meet the user's interests. The posts that appear first are those that have tags corresponding to tags in users' search history. The posts that follow are ordered from most recent to least recent.
User can view recipes from external websites directly on CrowdSauce	For posts that have recipes that link to an external website, users can open the recipes in a modal directly on the CrowdSauce website.
User can post comments on recipe posts	Users can post comments for a post they see in their feed in a comment field below the post.
User can "favorite" posts	Users can "favorite" or "unfavorite" posts that appear on their feed by clicking or un-clicking a button that resembles a heart on the post.
User can view trending posts	On a side bar on the feed page, users can see the titles of the most favorited recent posts on CrowdSauce.
User can view profile page	Users can view a profile page that displays their post history and allows them to visit their friends' profile pages.

User can receive email notifications for CrowdSauce activity	Users can receive emails alerting them when their posts are “favorited” or when their friends make posts. These emails contain a link that directs them to a quickview of a single post on the CrowdSauce website.
User can make changes to account in the preferences and settings page	Users can update their account details including name, email, and profile picture. In addition, they can subscribe or unsubscribe to email notifications.
User can keep track of a shopping list of ingredients	Users can add ingredients to their shopping list directly from the posts on their feed. This shopping list can then be viewed on a separate page in which items can be added or removed from it.

Figure 1 : Summary of user stories

Actor	**Task-level goal**	**Priority**
Recipe Sharer/Recipe Finder	Login through Facebook	1
Recipe Sharer/Recipe Finder	View friends list	4
Recipe Sharer/Recipe Finder	Customize app	2
Recipe Sharer/Recipe Finder	View account	3
Recipe Sharer	Share recipe with post	1
Recipe Sharer	Share custom recipe	2
Recipe Sharer	Share recipe from another website	2
Recipe Sharer	Share recipe with photos	1
Recipe Sharer	Share recipe with rating	2
Recipe Sharer	Share meal notes	2
Recipe Sharer	Share difficulty level of recipe	4
Recipe Sharer	Share preparation time of recipe	4
Recipe Sharer	View past posts	3
Recipe Sharer	Receive alerts when friends “favorite” shared post	4
Recipe Finder	View friends recipe posts	1
Recipe Finder	Search through posts	2
Recipe Finder	Comment on posts	1

Recipe Finder	“Favorite” posts	2
Recipe Finder	View most popular posts	4
Recipe Finder	Receive alerts when friends make post	4
Recipe Finder	Keep track of a shopping list for ingredients	4

Figure 2 : Actor-goal list of use cases

Architecture & Design

The frontend and backend components of this project were designed and developed independently of each other. The developers of the user interface considered the backend storage and processing as a black box. Similarly, the backend developers considered the user interface to be a black box during development. A RESTful API was added to streamline the communication between the frontend and backend.

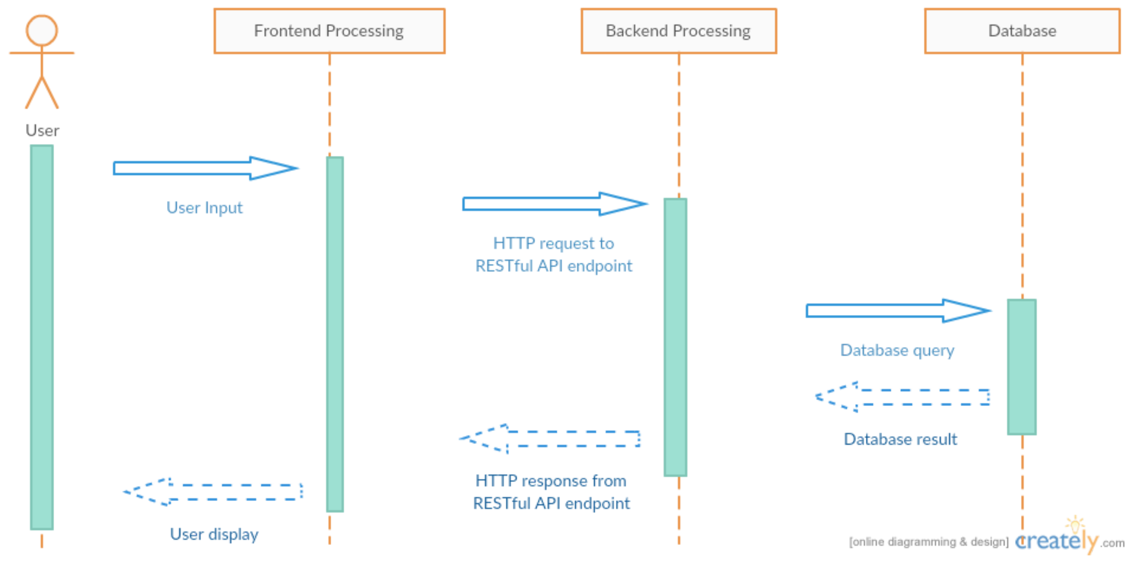
User
Frontend Processing
Backend Processing
Database
User Input
HTTP request to RESTful API endpoint
Database query
Database result
HTTP response from RESTful API endpoint
User display
[online diagramming & design] creately.com

Figure 3: Sequence diagram of high-level overview of dataflow

Database & Backend Processing

RethinkDB, a NoSQL database, was used to support our backend. Models and relationships among different models were set up to form the schema we used for storage.

The database structure is illustrated in Figure 4. We designed six models, and each row in a given model has certain fields that describe that particular data entry. We used “hasMany”, “hasOne”, and “belongsTo” relationships between the different models. We placed the following constraints when forming the schema:

- A user has one shopping list, and a shopping list belongs to one user.
- A user can have many posts, and a post belongs to one user.
- A post can be “favorited” by multiple users, and a user can “favorite” multiple posts.
- A post can have many tags, and a tag can be used by many posts.

The last two constraints were fulfilled by joining different models together. In order to place the "many-to-many" constraint for favorites between users (represented by the `Account` model) and posts (represented by the `Post` model), we added the Favorites model. Similarly, to impose that constraint for tags and posts, we added the `TagHistory` model. The database structure is shown in Figure 4.

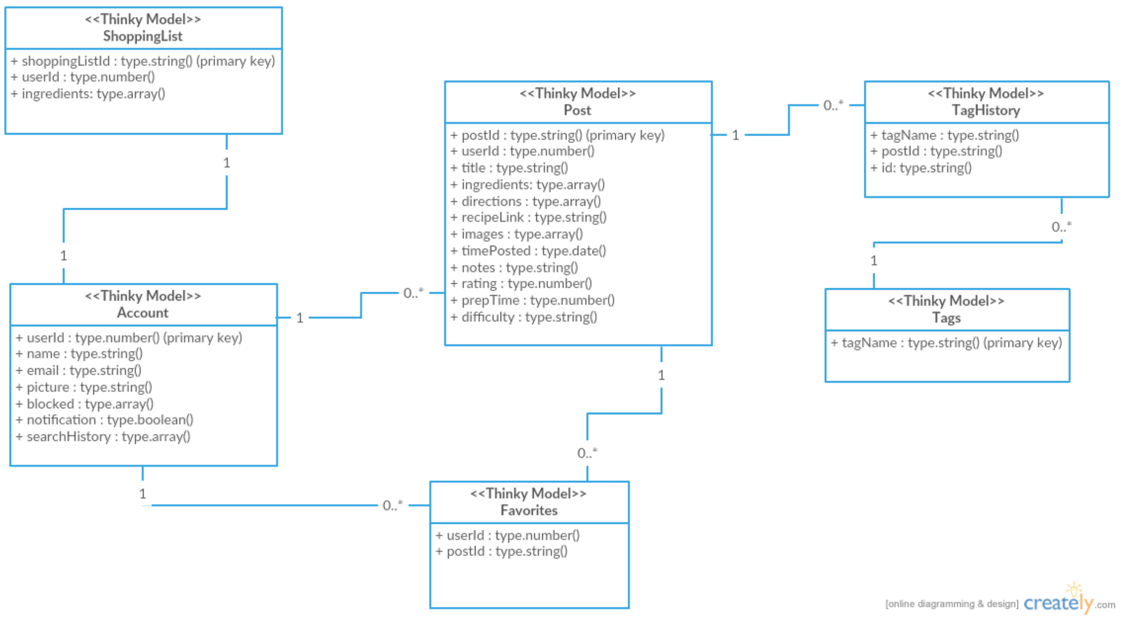
<<Thinky Model>>
ShoppingList
+ shoppingListId : type.string() (primary key)
+ userId : type.number()
+ ingredients: type.array()
<<Thinky Model>>
Post
+ postId : type.string() (primary key)
+ userId : type.number()
+ title : type.string()
+ ingredients: type.array()
+ directions : type.array()
+ recipeLink : type.string()
+ images : type.array()
+ timePosted : type.date()
+ notes : type.string()
+ rating : type.number()
+ prepTime : type.number()
+ difficulty : type.string()
<<Thinky Model>>
TagHistory
+ tagName : type.string()
+ postId : type.string()
+ id: type.string()
<<Thinky Model>>
Account
+ userId : type.number() (primary key)
+ name : type.string()
+ email : type.string()
+ picture : type.string()
+ blocked : type.array()
+ notification : type.boolean()
+ searchHistory : type.array()
<<Thinky Model>>
Tags
+ tagName : type.string() (primary key)
<<Thinky Model>>
Favorites
+ userId : type.number()
+ postId : type.string()
1
0..*
[online diagramming & design] creately.com

Figure 4: Database structure

The database is accessed and updated using RethinkDB queries and Thinky (a Node.js ORM for RethinkDB) in backend processing, which consists of handlers for each of the endpoints that our RESTful API supports.These endpoints support POST, GET, PUT, and DELETE requests. We have five sets of endpoints, and we implemented a class containing handlers for each set.

A router was implemented to direct HTTP requests to the proper handler. When a request is sent to one of the endpoints our RESTful API supports, the router decides which handler to direct the request to based on the URL the request is sent to. For instance, if a request is sent to a URL that begins with "api/accounts," then the router directs the request to the proper handler within `AccountHandler`. This flow is illustrated in Figure 5.

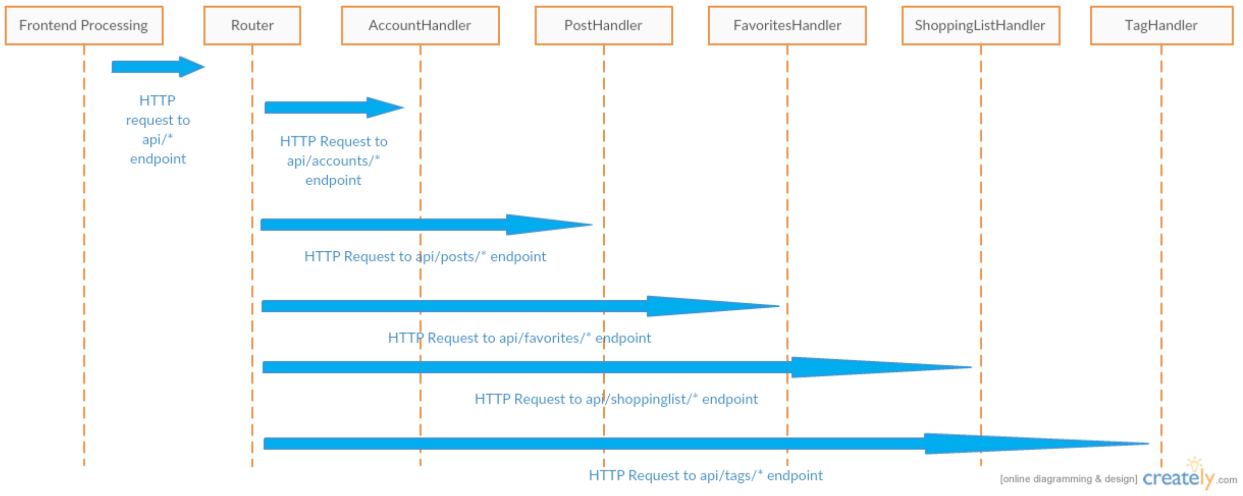
Frontend Processing
Router
AccountHandler
PostHandler
FavoritesHandler
ShoppingListHandler
TagHandler
HTTP request to api/* endpoint
HTTP Request to api/accounts/* endpoint
HTTP Request to api/posts/* endpoint
HTTP Request to api/favorites/* endpoint
HTTP Request to api/shoppinglist/* endpoint
HTTP Request to api/tags/* endpoint
[online diagramming & design] creately.com

Figure 5: High-level view of flow of HTTP requests to RESTful API endpoints

Each handler class implements functions that handle different kinds of requests. The various functions in each handler are highlighted in Figure 6.

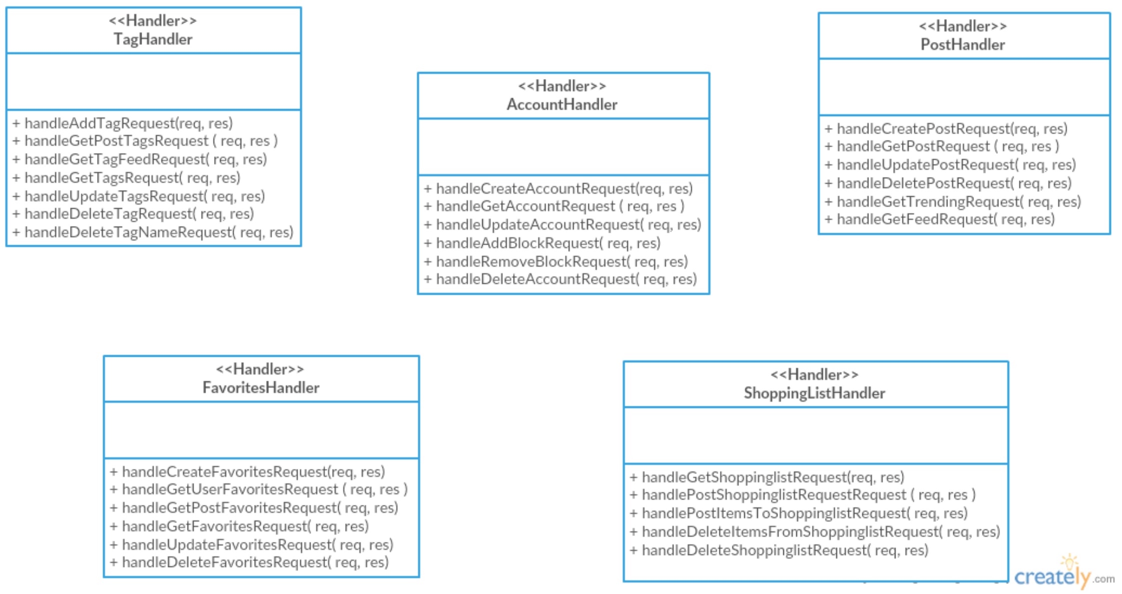
<<Handler>>
TagHandler
+ handleAddTagRequest(req, res)
+ handleGetPostTagsRequest (req, res)
+ handleGetTagFeedRequest(req, res)
+ handleGetTagsRequest(req, res)
+ handleUpdateTagsRequest(req, res)
+ handleDeleteTagRequest(req, res)
+ handleDeleteTagNameRequest(req, res)
<<Handler>>
AccountHandler
+ handleCreateAccountRequest(req, res)
+ handleGetAccountRequest (req, res)
+ handleUpdateAccountRequest(req, res)
+ handleAddBlockRequest(req, res)
+ handleRemoveBlockRequest(req, res)
+ handleDeleteAccountRequest(req, res)
<<Handler>>
PostHandler
+ handleCreatePostRequest(req, res)
+ handleGetPostRequest (req, res)
+ handleUpdatePostRequest(req, res)
+ handleDeletePostRequest(req, res)
+ handleGetTrendingRequest(req, res)
+ handleGetFeedRequest(req, res)
<<Handler>>
FavoritesHandler
+ handleCreateFavoritesRequest(req, res)
+ handleGetUserFavoritesRequest (req, res)
+ handleGetPostFavoritesRequest(req, res)
+ handleGetFavoritesRequest(req, res)
+ handleUpdateFavoritesRequest(req, res)
+ handleDeleteFavoritesRequest(req, res)
<<Handler>>
ShoppingListHandler
+ handleGetShoppinglistRequest(req, res)
+ handlePostShoppinglistRequestRequest (req, res)
+ handlePostItemsToShoppinglistRequest(req, res)
+ handleDeleteItemsFromShoppinglistRequest(req, res)
+ handleDeleteShoppinglistRequest(req, res)
creately.com

Figure 6: Class diagram of handlers

In general, the functions are named to describe the type of requests they handle. Functions with “create” or “add” in the name handle POST requests, functions with “get” in the name handle GET requests, functions with “update” in the name handle PUT requests, and functions with “delete” in the name handle DELETE requests. Each of the five sets of handlers interacts primarily with one model. For instance, all handlers in the `AccountHandler` class interact mainly with the `Account` model. There are some cases that a particular handler queries multiple models such as the handler that handles GET requests to the “api/tags/feed” endpoint which is expected to return a feed of posts with certain tags in them. This handler interacts with both the `TagHistory` model and the `Post` model. While handlers interact with multiple models, they do not interact with different handler classes.

Frontend & Interface Architecture

The Frontend team used React.js to build the CrowdSauce interface, as well as handle all front-end logic and data processing. A goal of the front end team from the beginning was to keep all front end code (almost entirely React.js classes) as modular as possible. We hoped to build singular components that contributed a specific view and/or functionality to the set of React classes.

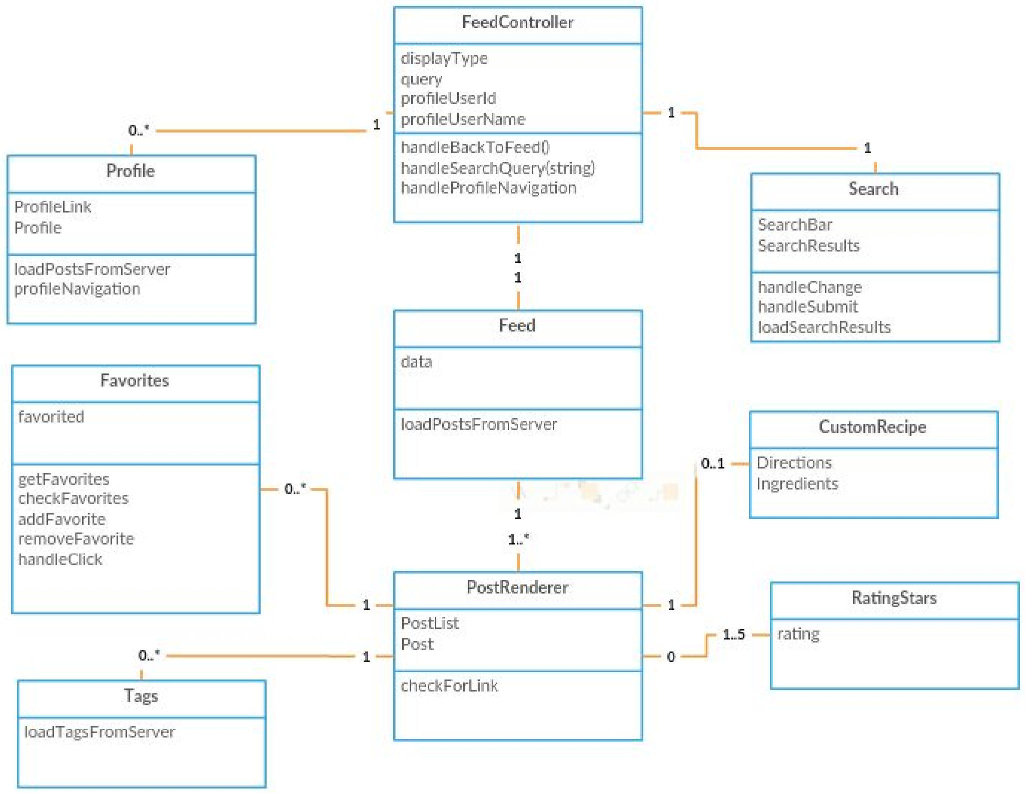

Figure 7: React Class structure

Above in Figure 7, we can see how the modular React classes were organized, and the dependencies that existed between classes. Due to React's use of a singular flow of data (using parent-child relationships), it was necessary that certain controller classes be built to manage the use of child classes. `FeedController`, for example, is the React class that manages and delegates all logic on the main feed page. It handles the changing of displays from the `Feed, Search, and Profile` classes. When an event occurs within `Search`, for example, that requires a change to `Feed` be made, the change is defined and handled by its parent class `FeedController.` The same parent-child logic is leveraged throughout the front end code to properly handle a correct data flow. To ensure that all logic is handled by only the necessary React classes, we also aimed to keep the `state and props` attributes of each React class as small as possible.

The display of recipe posts was managed through a `PostList`, which was made up of `Post` objects. Each `Post` object encapsulated behaviors such as editing, favoriting, and commenting. It also breaks down its display items such as `RatingStars`. `PostRenderer` holds both of these classes. `PostList` is employed by `FeedController`, which in turn uses it to render `Profile`, `Search`, `Tags`, `Feed`, and `Favorites`. `PostList` is a great example of abstracting out a UI element to a modular React class.

The posting of new recipes is managed by `PostRecipe`, which allows for both custom recipes and ones linked from outside websites. `PostRecipe` includes a previewer for images and the ability to list out ingredients and directions.

Each React component was able to access necessary data using RESTful API that the backend team had developed. Using the Swagger online documentation and API tool, the team was able to ensure that the correct endpoints existed and test their correctness. With a straight-forward API interface, the frontend team was able to focus mainly on organizing React components and rendering them successfully.

Reflections & Lessons Learned

The CrowdSauce team consisted of nine members who all contributed to the project in unique ways, so each member took away something different from the experience. Reflections from individual team members are given below.

Sheena Panthaplackel

During development, the team was divided into two sub-teams that focused on different aspects of the project. There was a frontend team and a backend team that worked independently. Working in smaller teams was effective in many ways because it narrowed down roles and responsibilities within the team. However, because these two groups worked independently, the communication between the sub-teams became strained. There were times when the backend team made changes without notifying the frontend team, which introduced several bugs on the frontend. In order to ease this communication gap, the backend team used the Swagger framework to clearly define the functionalities and parameters required for different endpoints. To add, there were times in the beginning when a particular sub-team noticed a bug in something implemented by the other sub-team. Poor communication between these groups prevented these bugs from being reported and fixed in a timely manner. To resolve this issue, we began bug reporting on GitHub Issues, which alerted everyone in the team of the major bugs.

Team members did not particularly move between sub-teams. This made people experts in the particular areas they worked in, which made development move a lot faster. In addition, because the same people worked in the same areas, the design and coding style remained consistent. However, because people did not move around, members of the backend team were unfamiliar were the frameworks and design used in the frontend, and similarly the members of the frontend team were unfamiliar with the frameworks and design used in the backend. The frontend team became dependent on the backend team and vice versa to debug issues they ran into.

Harry Jian

The most important reflection is perhaps that communication is key in a collaborated project such as this. Our team of nine split up into two smaller teams. The frontend team focused on designing the user interface and worked mainly with technologies such as ReactJS, HTML, JavaScript, and CSS, and the backend team focused on databases and building the RESTful API and worked mainly with RethinkDB and Thinky. At various points in our development process, the backend team changed the database models without informing the frontend team due to a lack of communication, and as a result the frontend team failed to reset their database, which caused numerous problems that would not have been present if the backend had communicated their changes on time. Also as a result of a lack of communication, collective code ownership suffered. However, we strived to fix our problems each iteration and worked on better communicating with each other on Slack. As a result I can say that our communication improved drastically over the course of the project.

Vishnu Indukuri

Working on larger teams is hard. Nearly all other teams that I've worked on have been 3-4 people. Somewhere between there and 8 member teams, the communication dynamic changes and you start to see the inefficiencies of larger teams. It becomes harder to communicate effectively and coordinate efforts. With four people or less, tasks are usually assigned to one person each but with larger teams, most tasks get two people assigned to them. I'm not quite sure why this happens because theoretically we should break the tasks into smaller pieces and then distribute each to a different person. I suppose due to the nature of real world tasks, they are usually harder to break up into clean unrelated subtasks so it is easier to have two people working on a larger task. Once more than one person is working on the same code, an entire host of problems arise in trying to keep each of our changes compatible with the other's in real time so that we don't have to look down the barrel of a large git merge conflict. I think while it was invaluable experience to have, anyone who wants to get a lot done in a small amount of time should stick to <5 person teams.

Charlie Martell

I had a number of takeaways while working on this project, notably, clean interface documentation between teams is vital, a high level of library familiarity is needed among everyone, and constant discussion among team members.

One thing I quickly picked up while working on CrowdSauce was between the backend and the frontend team there needed to be a better interface to show new features. Since the project was divided as it was the front-end team was developing front end assets for the backend features being implemented at the same time meaning the front-end did not always have a clear idea of what endpoints to call and what they would get back. To alleviate this problem I suggested we, the back-end team, work on a separate web page utilizing the Open API Initiatives tool for documenting RESTful APIs called Swagger. Swagger proved to be an invaluable tool as it provided a clean and clear interface that allowed the frontend to visually see all endpoints, what fields they take in as headers, queries and bodies, and what data schema they are returned back in. This made our work with the front-end go from chaotic to seamless in a week.

The other thing I realized during our initial weeks of this project was that everyone needed to have a really good understanding of even some of the smaller tools we integrated into our project. During our first weeks we had some trouble setting up endpoints with RethinkDB, our database engine, as it was looking like it was going to be a lot of code for each query from each endpoint. I suggested we use RethinkDB's proprietary ORM Thinky.io. This proved ultimately very helpful but took some time for all of us to understand the tools provided by the ORM and how we didn't need to manually make tables at all. After we were all intimately familiar with this tool we were able to make on-the-fly changes to our server with ease.

Finally working with a 'large' group as we did during this semester I learned a lot of things about communication among people. I realized that even though we all have busy schedules outside the scope of this class we all needed to be available at all times since we were all working on different moving parts and if someone needed help working with a tool someone else wrote and that person was unable to speak to them at that time to assist them it would mean their time would be wasted trying to understand something that could have been quickly explained.

Overall this was a really great experience getting to work with many college level developers on this project and I feel like I am coming away not only as a better Javascript developer but as a Software Developer as a whole.

Li Jen Tu

Starting a project from scratch was more difficult than I imagined. Not everyone was familiar with the development tools (ReactJS, Node, rethinkdb, etc) used compared to last semester where everyone was familiar with Java. Though each tool alone may be simple, integrating each tool with one another to solve our user stories was the difficult part. Our team chose to split into the front end group and back end group. This proved to be beneficial since we were able to focus on learning some tools thoroughly instead of dabbling in all the tools. The bad side to this split was that both groups did not know how the other codes worked. This could be quite frustrating when things break or bugs appear and we were not sure how to fix it. Swagger did help alleviate a lot of the problems as it provided a clear visualization of the endpoints.

Working in a large group also presented similar challenges. Finding time where all of us could meet was difficult. To counter this, We used Slack and Github Issues which help immensely with communication. Another challenge with working in large groups were merge conflicts. It was nerve wracking when group members were working on the same files. In order to avoid this as much as possible, I learned to constantly push and pull the code.

Overall, this project was an enlightening experience for me. I got to learn a lot about JavaScript, Node, React, and the Facebook API. In particular, it was very interesting to see how different node is from php. Also, this project helped me realize the importance of communication in large groups and the problems that may arise. All in all, this was a great learning experience.

Luke Puchner-Hardman

I was surprised at the amount of work that goes into the development of what is, essentially, just a blog for food. Although unlike most projects I've worked on, the difficulty was not in writing algorithms or designing data structures but rather simply learning the tools that we were using. I'm sure a team with a decent amount of experience in NodeJS and RethinkDB would have found the project to be very easy, but for us it was a good learning experience.

The team was split into frontend and backend (I was on backend) and we did not switch around much so few had to learn how both ends worked. The general pattern for the two week iterations was that we would spend the first week assigning tasks to each of the team members and the second week working on those tasks. Unlike last semester, we mostly worked on our own instead of in pairs or groups. I think this was good for a school project because it meant we were not limited by when our schedules matched up and we did not have to write anything algorithmically complex that would have needed a second pair of eyes. We developed in such a way that the frontend and backend could mostly work independently and we could link it all together near the end of the second week.

Overall, I still do not like web development, mostly because everything is dynamically typed and I feel like I spent at least as much time googling as writing code.

Charlie Anderson

Looking back, one of the biggest issues we faced was communication among such a large team. We handled this through multiple weekly meetings as well as emails and constant communication through slack.com. Having 9 members on our team made it twice the size of a normal team in my experience. We managed to break up the team by splitting between frontend and backend sub-teams to specialize in one of the two areas. This allowed us to more easily designate tasks which were evenly split into frontend and backend tasks. I ended up on the front-end team, and I felt it helped a lot to focus in one area as opposed to trying to learn the full stack in a limited amount of time.

We also had the issue of handling and learning a lot of new technology in a relatively short period of time. We used several different tools and frameworks such as Rethinkdb, Swagger, React.js, Node.js and more, which a lot of people had never worked with before. I think we successfully handled this though, as splitting the team between frontend and backend helped to cut down on how much we each needed to know. It also helped make people more of experts in a particular field, making it easy to find someone to get help if a particular issue needed solving. Plus, if an issue outside of someone's expertise needed to be fixed, we used the Github Issues feature to quickly ask other members for help.

In the end I think we successfully handled the situation and came out learning a lot from our experience. I feel like I not only gained web development skills, but also learned how to manage my time with a large group of people. The communication skills I gained will undoubtedly be helpful to me in the future, regardless of where my career goes.

Jonathan Alkalai

The main takeaway that I would have after working on CrowdSauce for the semester is how important organization and consistent communication are to successful project development. Every member of our 9-team group was a talented programmer, capable of tackling each iteration's work. But the main bottleneck of our development was having a clear understanding of who was tackling certain issues, who to contact with questions, and the exact details of what should be built. To help with our team communication we relied on both Slack and GitHub Issues. While the tools helped, there were often points of confusion and uncertainty in our development. I think our team could have benefited greatly from someone assuming a role similar to a Project Manager. With a dedicated leader and organizer, I think that we all could have been more effective and more often on the same page.

From a technical standpoint, I am very happy with the tools that we chose to build our web application with (Node.js, React.js, RethinkDB), and how many team members responded to the challenge of learning new tools. While there was a clear learning curve, it seems like the team enjoyed learning the new tools and getting familiar with their usage in various other web applications. I personally had a great experience with React.js, and I hope to use the tool more often in the future.

John O'Keefe

My main takeaway from my experience working on CrowdSauce is that upfront project design pays off well down the road. I worked on the front end team throughout the project and witnessed many times where fore-planning would have avoided our major pitfalls. For example, we ran into issues with being able to access the Facebook user ID and server access token of the website's current user, items needed for making calls to our API. The issues were caused by the fact that access to the Facebook details was enabled through a normal JavaScript, rather than through our React system. This disconnect between React and JavaScript likely could have been avoided if we had found a solution for managing external API access before beginning development.

Also, upfront design could have helped alleviate communication problems we encountered throughout the semester. If the group had decided more firmly on the class structures and libraries we would use at the start of the semester, we could have avoided problems brought on by the introduction of new libraries and classes later in the development process. Many of the issues raised on GitHub stemmed from confusion caused by the frontend team not understanding the class hierarchies or libraries of the backend team, or vice versa. In the end, the project was a great experience in front end web development, and I learned a lot about the value of different development strategies.